Bachelor's Thesis

Information Technology

2014

Opeyemi Michael Ajayi

ARCHITECTURE PERSPECTIVE OF NOSQL

- User Experience and Scalability of Cassandra and MongoDB.

TURUN AMMATTIKORKEAKOULU
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information Technology | Internet Technology

2014 | 72

Instructor: Almurrani Balsam

Opeyemi Michael Ajayi

ARCHITECTURE PERSPECTIVE OF NOSQL: USER EXPERIENCE AND SCALABILITY OF CASSANDRA AND MONGODB

The enormous amount of data stored today by web applications and non-web applications alike are becoming alarming as a result of slow development of storage technologies required to cope with the scale and agility challenges. This is very typical of highly ordered data that needed to be accessed by multiple applications concurrently. As developers and database administrators made effort to cope with users data, new technologies emerged to solve the limitations of traditional database designs and provide more features to enhance scalability and user exxperience.

This thesis examined the traditional relational database management systems and NoSQL database technology. The aim of this work was to test the two technologies by comparing a relational database with two NoSQL databases and analyze how NoSQL technology is providing solutions to the scalabilty and performance limitations of relational databases.

As a result of this thesis, a relational database and two NoSQL databases were implemented to hold the data of a Twitter clone application. The results of the comparison revealed that MongoDB can better handle complex queries than MySQL because of its simple and flexible schema. MySQL on the other hand would better serve in simple search queries because of its logical data structure. Cassandra exhibit the most data availability due to data partitioning over sets of nodes.

KEYWORDS:

NoSQL, MySQL, MongoDB, Cassandra

FOREWARD

The list of individuals that have contributed to the success of this work goes on and on. Nevertheless, I would like to thank Mr. Almurrani Balsam for sparing time of his busy schedule to supervise this thesis. I would like to extend my gratitude to Mr. Ashaolu Paul for explaining some of the unclear concepts in databases. This knowledge proved priceless to the success of this project work.

Finally, I would like to thank Mrs. Ana Kupri for her motivation in the course of this thesis work.

2014, Turku

Opeyemi Michael Ajayi

CONTENTS

FIGURES

CHARTS

TABLES

LIST OF ABBREVIATIONS AND SYMBOLS

ACID	Atomicity, Consistency, Isolation, Durability
Atomicity	A relational database property that guarantees that either all units of a transaction are carried out or none are.
Durability	A relational database property that ensures a transaction cannot be lost after being committed to the database.
Consistency	A relational database property that ensures a database remains in a consistent state before and after a transaction.
Dependency	a constraint between two attributes in a database table
Isolation	A relational database property that ensures multiple transactions operating on the same data do not interfere with each other.
Nodes	i. a server in a cluster ii. item of data that can be accessed by multiple routes.
Normalization	A process of organizing a database table to eliminate redundancy
NoSQL	(generally interpreted as “Not only SQL ”) : a class of database management systems that does not use SQL for data manipulation.

Replication	Server : cloning of server to enhance reliability. Data: duplicating and storing multiple data across clusters to enhance availability and support disaster recovery.
Scalability	Ability of a system to adapt to an expanding workload.
Sharding	Storing data on multiple machines.
SQL	A standardized query language for requesting information from a database.
UX	User Experience

1 INTRODUCTION

1.1 Background

For years, individuals and organizations have used relational database to store what is known as structured data. The data is sub-divided into groups called tables. The tables store well-defined units of data in terms of type, size and other database value restriction rules known as constraints. Each unit of data is called a column while each unit of the group is called a row. The columns can have relationships, such as parent-child relationship, defined across themselves, and therefore the name relational database. However, since consistency is a vital factor, horizontal scaling has been a challenging, and more or less an impossible task. [1]

Recently, with the increase in number of large web applications, researches have been conducted to discovering alternative options of handling data at scale. Hence, developers have emerged from using only Relational databases to exploring non-relational options, such as schemaless data structures, simple replication, high availability, horizontal scaling, and new querying methods. These newly discovered options are collectively referred to as NoSQL databases. [2]

1.1.1 What NoSQL is and what it is not

NoSQL is a general term that refers to any data store that does not comply with the well known and established traditional relational model (Gaurav Vaish, 2013). In order words, the data is non-relational and does not use SQL as the data query language. NoSQL is also used to describe the databases that solve the issues of scalability and availability against that of atomicity and consistency in relational database.

It is however worth mentioning that NoSQL is not itself a single database or a single technology but a class of databases and a group of diverse and

sometimes related databases. In *Chapter 3, NoSQL Databases*, author explores various genres of database types available under NoSQL.

1.2 Research Aims and Objectives

The aim of the current research work is to demonstrate the architecture of NoSQL technologies in contrast to the traditional relational database. The research questions can be framed as: How have NoSQL technologies solved the scale and agility challenges facing modern applications and taken advantage of the cheap storage and processing power available today?

The objectives of this work is to:

- examine the user experience and scalability of two NoSQL databases namely Apache Cassandra and MongoDB.
- implement these two databases to demonstrate the architecture of NoSQL technologies.

1.3 Thesis Overview

This thesis work is organized into 6 different chapters, each one addressing specific aspects of the project. The content of each chapter is as follows:

Chapter 1: Introduction

The introductory chapter presents the overview of the thesis. It gives the background information and explains the aims, objectives and motivation for carrying out the research work.

Chapter 2: Literature Review

This chapter encompasses the fundamental concepts of databases. Through this chapter, audience can familiarize with early database models; Navigational, Object-oriented and relational database models and comprehend the differences between these models.

Chapter 3: NoSQL Databases

In this chapter, author will discuss NoSQL databases, what led to their discovery, the different types and examples. Finally, the chapter will help readers have insight into which database best suit their data store and refinement.

Chapter 4: MongoDB

This chapter will demonstrate the strength of NoSQL technology over relational database using Create, Read and Delete operations to perform a comparison between MongoDB, a document-oriented and open-source database management system, and MySQL, a relational and open-source database management system. The two databases will be holding the data of a social network. The audience will understand the architecture, user experience and scalability of MongoDB database management system.

Chapter 5 : Apache Cassandra

This chapter will demonstrate the strength of NoSQL technology over traditional relational database using Create, Read and Delete operations to perform a comparison between Apache Cassandra, a column-oriented and open-source distributed database management system, and MySQL. The audience will understand the architecture, user experience and scalaility of Apache Cassandra database management system.

Chapter 6: Conclusion

This chapter concludes the research by providing the goals of the research and suggesting possible ways of further improving it.

2 LITERATURE REVIEW

2.1 Data

In computing and data processing, data are distinct information usually translated into special forms that are convenient to process. Data can be in form of characters, symbols or signals on which operations are performed by a computer. They can be structured graphically with a set of connected nodes, or as a tree with nodes having parent-child relationship, or as a table with rows and columns. Moreover, in database management systems, data files are files in which the database information are stored. [3] [4]

2.2 Databases

Databases are collections of information organized to provide efficient retrieval of data [5]. Although the term database is commonly used to refer to the entire database system, however, it actually refers only to the collection and the data. The system that handles the storage, modification and extraction of information from a database is the Database Management System (DBMS).

In early designs and implementations, linked lists were used to create relations between data and to locate specific data. However, these models were not standardize and extensive training were required to use them efficiently. These models are explained briefly below. [6]

2.3 Navigational and Object-Oriented Databases

Besides relational and NoSQL are two other database models that were developed in the past. Although Navigational and object-oriented databases are not as successful as the relational and NoSQL databases, nevertheless, they contribute greatly to database evolution.

2.3.1 Navigational Databases

Navigational databases belong to the first generation databases. In this type of databases, objects or records are found mainly by tracing references from one object to another. This is opposite to the relational model which utilizes a declarative technique that queries a database for a record rather than navigate to it. The main drawback of navigational databases is that users need to be quite familiar with the fundamental physical structure of the database to guery for data. Moreover, adding more field to a database would mean reconstructing the whole storage design. Similarly, choosing a suitable implementation was a problem due to lack of standardization among vendors. [6] [7]

2.3.2 Object-oriented Databases

Object-oriented databases also known as object databases were first developed in the 1980s and since then have been an essential part of database evolution. They are mostly used with object-oriented programming languages in object-oriented field. An object-oriented database management system supports modelling and creating data as objects. This characteristic permits object-oriented programmers to develop new applications by cloning or modifying existing ones within the database management system. The major limitation of this model is that modifying a schema of an Object DBMS application means the entire database will be recompiled. [8] [9]

2.4 Relational Databases

Relational databases organize data into tables of rows and columns. Each row represents a record and each column represents a field. Tables are linked with each other based on defined relationships such as foreign keys or common columns. These relationships enable user to retrieve and join data from one or several tables using a single query. Abstractly, tables represent entities such as artists, movies or roles, which become handy in designing the database schema as actual objects need to be mapped to the database in addition with the

relations between them. Figure 2.1 below represents the design of a typical database schema.

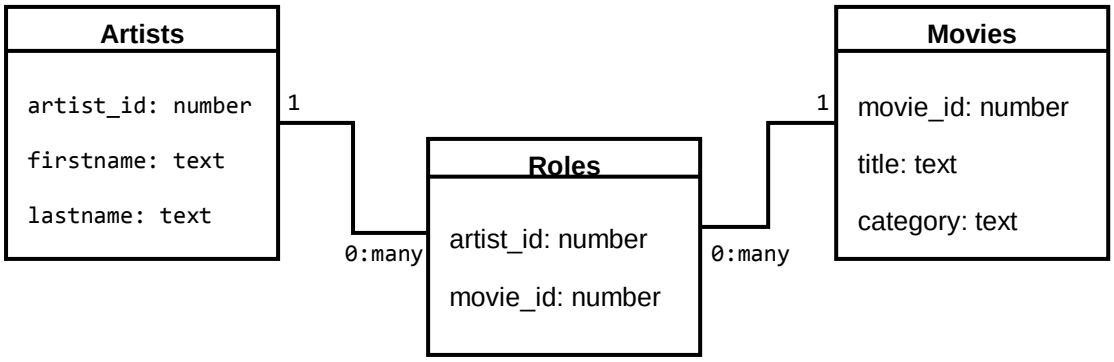

Figure 2.1. An example of a database schema

2.4.1 Normalization

A significant aspect of designing a relational database is ensuring the schema is normalized. Normalization is the process of classifying and organizing data in a database. The two goals of the process is to protect data and make the database more flexible by eliminating data redundancy and inconsistent dependency. These can be achieved by establishing relationship between tables of the database according to the rules described below:

1. First Normal Form (1NF): Usually represented as 1NF in practical applications, First Normal Form sets the fundamental guidelines for an organized database as follows:
    - Eliminate groups of duplicate data by creating a separate table for each group of related data.
    - identify each group of related data with a primary key.
2. Second Normal Form (2NF): Also written as 2NF, Second Normal Form further emphasizes the idea of eliminating duplicate data as follows:
    - satisfy all the requirements of First Normal Form.
    - if a set of values apply to multiple records, create separate tables for them.

- relate these tables with a foreign key.

3. Third Normal Form (3NF): Denoted as 3NF in practical application, third Normal Form sets the following rules:
    - Meet all the required conditions of Second Normal Form.
    - Eliminate fields that do not depend on the primary key of a table and put them into another table if necessary.
4. Boyce-Codd Normal Form (BCNF or 3.5NF): The Boyce-Codd Normal Form is also called third and half (3.5) Normal Form. It appends one more rule to the 3NF to address an anomality not treated by 3NF:
    - Meet the requirements of Third Normal Form.
    - Every determinant must be either a primary key or a candidate key.
5. Fourth Normal Form (3NF): written as 4NF in practical application, Fourth Normal Form sets the following rules:
    - satisfy all the requirements of 3NF.
    - two or more multi-valued attributes should not be included in the same relation.
6. Fifth Normal Form (3NF): Also refered to as 5NF, Fifth Normal Form sets the following rules:
    - Meet all the required conditions of 4NF.
    - Every non-trivial join dependency in a table is implied by the candidate keys.

It is worth mentioning that for a database schema to satisfy 2NF, it must first satisfy 1NF and to satisfy 3NF, it must first satisfy 2NF. The fourth and fifth normal forms are seldom considered in practical design. Therefore, a database schema is considered normalized if it satisfies the first three normal forms. [10] [11] [12]

2.4.2 ACID Properties of Relational Databases

The four key characteristics of a relational database transaction that guarantee its reliability (data consistency and integrity) is referred to as ACID properties.

Traditional RDBMS applications have focused on ACID transactions (Vaish, 2013). The term ACID is an acronym for Atomicity, Consistency, Isolation, and Durability. Each characteristic is explained below in the context of databases.

- Atomicity

  The atomicity of a database ensures that either all units or steps of a transaction are carried out or none are. In other words, if one unit of a database transaction fails, then the changes made to other units are rolled back to the original states they were before the transaction started.

- Consistency

  The consistency characteristic of a database guarantees that a database remains in a consistent state before and after a transaction, irrespective of wether the transaction fails or not. For example, if an error occurred in the process of account X transferring fund to account Y, the system will automatically rollback the completed part of the transaction so that fund is not deducted from account X and fund is not added to account Y. However, the successful completion of the transaction will mean all steps have been properly executed and the system will be in a valid state.

- Isolation

  The isolation property of a database guarantees that a running transaction is isolated from another transaction performing similar task and running simultnously, such that it appears that no other action is being carried out by the system. In other words, transactions operating on the same data do not interfere with each other.

- Durability

  The durability of a database guarantees that a successfully completed transaction is committed permanently to the database and changes cannot be lost afterwards. Considering the fund transfer from account X to Account Y, once the system confirms that account Y has been credited, the changes persist even if the system crashes. [13] [14] [15]

However essential these properties have proven to be, they are quite inconsistent with availability and performance demanded by today web applications.

2.4.3 Advantages of Relational Databases

Over the years, relational database management systems have offered quite robust information management tools to developers and businesses. The following are some of the merits of relational database model:

- Data Structure
  The system and its tabular form is easy for users to comprehend and use due to its natural structure and organization of data for accessibility. The database structured queries can search for matching entries in any column of the database. [16]

- Multiple User Access
  Relational database management systems permit more than one users to concurrently access the same database. This is made possible through an inbuilt functionality that locks and manages transaction as data is being modified. It prevents users operations from colliding and also from accessing partly updated records. [16]

- Authentications and Privileges
  Relational database management systems have authentication and privilege control features that permit database administrators to limit database access to only authorized users. Moreover, administrators can also grant access on the basis of the task the user wants to perform. [16]

- Network Access
  In relational database management systems, users can access and use the database without logging in through the physical computer system. The database management system uses server deamon, a special software program that listens for requests on a network, to connect

clients to the database. This becomes an additional security layer for the database and provides convinence for the clients. [16]

- Speed
  Some database models such as the NoSQL (non-relational) model may be considered faster than the relational database model, however, the advantageous ease of use of relational model makes the slower speed a reasonable trade-off. In addition, relational database management systems have in-built optimizations and robust designs that improve speed of handling most datasets and applications. [16]

- Maintenance
  Relational database management systems are built with handy maintenance tools that database administrators can use to easily maintain, test, repair and back up the databases in the system. [16]

- Language
  Relational database management systems support a standardized interaction and querying language known as Structured Query Language (SQL), for requesting information from a database. The language uses standard English language keywords, making the syntax simple and intuitive for database administrators to learn.[16]

2.4.4 Limitations of Relational Databases

Despite the merits of relational database model, the technology also poses limitations which are not only evident within the database, but also in the mechanisms through which applications access the data. Some major limitations of relational database model are explained below:

- Data Complexity
  A relational database management system does not oblige database designers to enforce a coherent table structure. As much as this flexibility may be beneficial for skilled designers, inexperienced ones may design

systems that cause unnecessary complexity or restrict the database from future development due to poorly selected data types. Such loophole poses risk on a company's data. [17]

- Broken Keys and Records

  In relational databases, shared keys are needed to connect information located across multiple tables. For instance, an artist table may store actors and actresses information, with a unique index number (key) identifying the record of a particular actor within the table. A role table may identify the actor only by that index number. Consequently, if the datatypes connecting the keys are different, the database will need a modification by the designer. Similarly, a table without a unique key may cause the database to output inaccurate result when querried. In addition, a user could accidentally corrupt data and break records if the application accessing a database is not programmed to lock records during an update. [17]

- Administrator Expertise

  The skill required by a relational database administrator is directly proportional to the complexity of the database. A complex database may require more than the skill of a small business database administrator. Moreover, if an administrator does not steadily involve in best practice design, a successor may not comprehend the hidden complication that could result in broken queries or inaccurate results. [17]

- Hardware Performance

  Complex SQL queries such as JOINs to collate data across multiple tables increase development time and therefore require sophisticated processing power. While most personal computers might be able to manage the size and complexity of small application databases, a database with very complex data structure or external data sources may demand more powerful servers to process queries and return results within a satisfactory response time. [17]

3 NOSQL DATABASES

The NoSQL trend began in the early years of the 2000's when companies and organizations started researching and investing into distributed database. By distributed database, we refer to a database that can scale to manage millions of users and billions of connected mobile, smartphone, internet TV and many more devices. This gave rise to different categories of NoSQL databases with each better serving in specific situations over others.

3.1 Why NoSQL?

Beyond solving scalability issues, NoSQL databases offer several other benefits including the following:

3.1.1 Schemaless Data Representation

Being schemaless means a database allows the storing of any type of data without any prior definition of how the data are organized in the database. As a result, a database designer is able to evolve a data structure over time. Such evolution may include adding new fields or columns, or nesting one data into another, without causing service interruption in the case of making important application changes in real-time.

3.1.2 Uninterrupted Data Availability

Companies and developers cannot afford downtime because of high competition in the marketplace and the havoc it could cause their reputation. NoSQL databases are developed with a distributed architecture to prevent any point of failure. If one or few servers, also called nodes fail, the system can continue operation with other servers without losing data, by that exhibiting fault tolerance. This is greatly advantageous as database administrators can perform update operations without necessarily taking the database offline.

3.1.3 Location Independence

By location independence, we mean the ability to perform read and and write operations irrespective of where the operations were carried out, and to have any write functionality take effect from the location, so that users and machines from other sites can also access it. In relational databases, techniques such as master/slave architecture can sometimes be used to achieve location independent read operations, however, this is not the same for write operations especially when it involve large volume of data. Local independence is beneficial in several other scenarios such as serving customers in different geographical locations and localizing data at those sites for quick access.

3.1.4 New Transactional Capabilities

In recent times, the “Consistency” property in ACID transaction have been proven not required in database driven systems. However, this is not to jeopardize data, but to relate to how modern applications guarantee consistency across widely distributed systems. NoSQL databases do not utilize the type of consistency in relational database management systems because there are no JOIN operations which require strict rule of consistency. Rather, the consistency in NoSQL databases transaction involves an instant or conditional consistency of data across all servers participating in a distributed database. Nevertheless, the data is still safe and quarantees the Atomicity, Integrity and Durabilty properties of the RDBMS ACID.

3.1.5 Quality Architecture

NoSQL databases provide suitable architectures for specific application. It is crucial that organizations and developers adopt a platform that best suits and keeps their large volume of data in the context of their applications. Several NoSQL databases provide quality architecture that can handle the type of applications that demand high scalability, data distribution and uninterrupted availability.

3.1.6 Analytics and Business Intelligence

Another essential and strategic reason an organization should employ a NoSQL database is the ability to mine large volume of data for insights that could give its business a competitive edge. Such extraction of business intelligence is a difficult task to perform with a relational database management system, especially when it involves large volume of data. Besides providing storage and managing business application data, NoSQL database management systems also analyze and provide quick understanding of complex data set, as well as facilitate decision-making. [18]

3.2 Types of NoSQL Databases

An essential aspect of NoSQL databases is that most of them are open source and community induced. In addition, they are categorized base on how they store data. Table 3.1 below shows the categories of NoSQL databases and lists examples of each.

Table 3.1 NoSQL database categories and examples

Key-Value	Column-Oriented	Document Store	Graph Store
Riak	Cassandra	MongoDB	Neo4J
Redis	Hypertable	CouchDB	InfiniteGraph
MemcacheDB	Hbase/Hadoop	Terrastore	FlockDB
Membase	SimpleDB	RavenDB	
Voldemort	Cloudera		

The list in the table above is not by any means exhaustive as more and more offerings are coming into the market. Next below are brief explanations of the database types mentioned above:

3.2.1 Key-Value (KV) Store

Key value databases store data as values, and pair each value with a key the same way as hash table. In addition, some KV database implementations allow a key to have collection of values but ths is not necessary. Similar to document stores, a schema does not need to be imposed on the value. However, the two models differ in two ways. First, while a document store can create a key when a new document is inserted, a key-value store requires that the key is specified. Second, in document store, a value can be indexed and queried but a key-value store requires that the key of a value be supplied to retrieve the value. Key-value stores like Redis support various value types namely; strings, lists, hashes, sets, and sorted sets.

Below are few examples of a basic data operation using Redis:

```
//SET – add string to collection
SET artist “artist list”

//Hash – sets field value
HSET artist firstName “Jason”

//Hash – sets field value
HSET artist lastName “Statham”

//Set – creates/updates
SADD “Jason:followers” “L1” “L2”

//Set – creates/updates
SADD “Ryan:followers” “L2” “L1”

//Intersection of followers
SINTER “Jason:followers” “Ryan:followers”

//Union of followers
```

```
SUNION "Jason:followers" "Ryan:followers"
```

Key-value stores are most prominent for querying against keys. Sometimes, it is possible to cleverly generate the keys and query against arrays of keys. Redis for instance, permits the retrieval of a list of all the keys matching a glob-style pattern. [1, p. 41]

3.2.2 Column-oriented

In column-oriented or columnar databases, data is stored as section of columns as opposed to the two-dimensional tables in RDBMS where data is displayed in rows and columns. As irrelevant this difference may seem, adding columns in column-oriented databases is cheap and done on a row-by-row basis. Assuming we want to store the data in table 3.2 below:

Table 3.2 Column-oriented database example

Artist_Id	Firstname	Lastname	Age	Country
A01	Jason	Statham	45	England
A02	Clark	Gregg	47	America
A03	Meg	Ryan	50	America
A04	David	Kross	39	Germany

The data in RDBMS may be serial and stored as:

Table 3.3 Data representation in RDBMS

A01	Jason	Statham	45	England
A02	Clark	Gregg	47	America
A03	Meg	Ryan	50	America
A04	David	Kross	39	Germany

But in column-oriented databases, it will be stored internally as:

Table 3.4 Data representation in Column-oriented databases

A01	A02	A03	A04
Jason	Clark	Meg	David
Statham	Gregg	Ryan	Kross
45	47	50	39
England	America	America	Germany

The advantage of column-oriented databases is that user can add new columns in the future without the need to supply default values for existing rows for the new columns. This gives a flexible model and unique design allowing users to consider new columns in future for unanticipated scenarios and requirements. [1, p. 26]

3.2.3 Document Store

Document store, also referred to as document-oriented databases, store each record and its associated data as a document. Majority of the databases under this umbrella use JSON, XML, BSON or YAML as data-interchange formats. Unlike relational databases, document store databases do not need to have their structure specified in advance, hence are said to be semi-structured. This gives a high degree of flexibility on database schema by allowing data to be more logically and naturally grouped together. For instance, two records may have entirely different set of fields. However, indexes can be created and queried. Below are few instances of document content using JSON:

A document may include an artist whose entire details are not known:

```
{
```

```
  "Artist_Id": "A01",
  "firstname": "Jason",
  "lastname" : "Statham",
  "Age"      : 45,
  "Country"  : "England"
}
```

Another document may provide entire details about another artist:

```
{
  "Artist_Id": "A01",
  "firstname": "Jason",
  "lastname" : "Statham",
  "Age"      : 45,
  "Country"  : "England",
  "otherInfo": "Travelling",
  "Movies"   : [
     "Transporter",
     "Transporter 3"
  ]
}
```

A third document may contain information about one of the movies:

```
{ "movieCode"   : "M01",
  "title"       : "transporter 3",
  "Category"    : "action",
  "Description" : "Frank Martins puts on the driving
                   gloves"  }
```

In the above instances, the first two documents are quite similar with the second document more detailed than first. However, the content of the third document is not correlated with the first two in any way since it is about a movie and not an artist. The schema flexibility of document-oriented databases has made them more popularly implemented and used.

A noticeable advantage, as obvious in the above instances, is that content schema is dynamic or loosely defined. This is very handy in web applications where storing variety of content may surface in future. In addition, performing search across multiple entities becomes easier than in relational database management systems or even columnar databases because one can query directly across the entire database. [1, p. 29]

3.2.4 Graph Store

Graph databases are special NoSQL databases that handle highly interconnected data called nodes. The relationships between the nodes are represented as graphs. Two nodes in a graph can have many links representing the multiple replationship between them, such as network topologies between connected workstations, social relationship between people, or transport links between places. An important quality of graph databases that distinguishes them from relational databases is that each element has a pointer to its neighboring element and therefore does not need to index every element. This quality is referred to as index-free adjacency.

Below is an example of what a graph representation may look like:

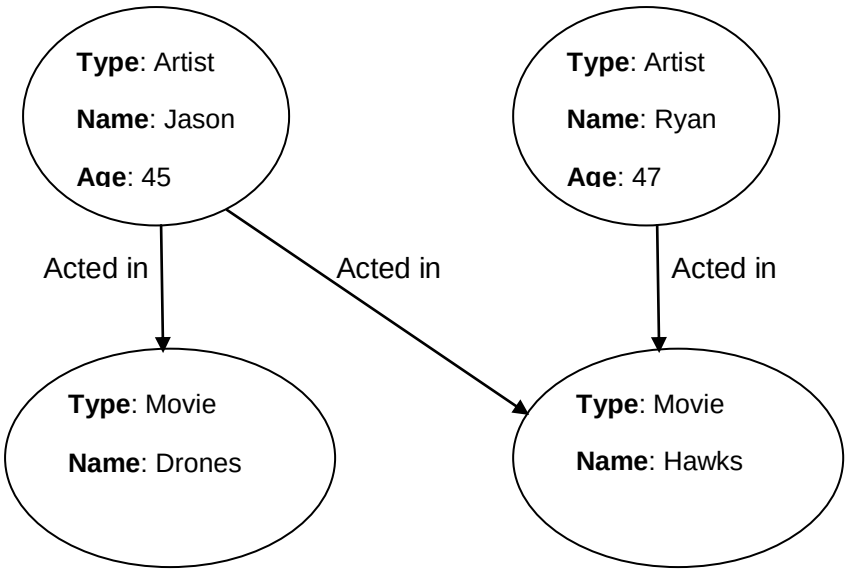

Figure 3.1. A basic graph store use case

Graph databases are best known for serving special purpose of handling relation-heavy data such as social network users. They enhance smooth representation, retrieval and manipulation of relationship between the entities in the system. In fact, without relationships among the entities, there will be no use case for graph databases. In view of this, a social networking application may want to store data in a document store while relationships are stored in graph databases. [1, p. 43]

3.3 Database Replication

Database replication is the deployment of multiple servers called replicas. Replication is used in NoSQL databases to provide high availability, reliability and performance without the need of separate applications to handle the tasks. MongoDB for example uses a replication configuration in which a server serves as primary replica and other servers as secondary. The function of the primary replica is to manage and log all write operations in a separate group where the secondary replicas can read and apply them. The load on a primary replica is often reduced with the ability of the secondary replicas to read from another secondary replica. Figure 3.4 below depicts this process. As much as having a duplicate copy of data provides reliability and huge performance, it also poses the chance of the data not being the most updated.

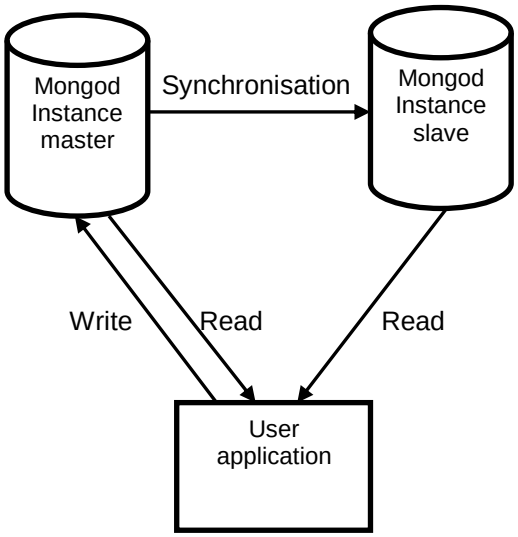

Figure 3.2. MongoDB Replica deployment and usage

In situations where the primary replica malfunctions and goes offline, the system resorts to voting a secondary replica to replace the primary. Should the voting results to draw among the secondary replicas due to even number of secondary replicas, an arbiter server then provides a vote in the election. An arbiter server exists for this purpose only. [6, p. 7]

3.4 Database Auto-sharding

Auto-sharding is the term that describes how NoSQL databases natively and automatically spread data across multiple servers without involving applications participation. This means applications activities are not disrupted nor do applications have to be down before servers can be added or removed from the data layer. As a result, performance increases as each server handles separate sets of data. However, because replication provides improved performance as well as reliability, it is recommended over sharding. [6, p. 8]

4 MONGODB

MongoDB is a document-store database designed for scalability, high availability and performance. It allows data persistence in a nested state and have the ability to query the nested data in an undefined fashion. In addition, it does not impose schema, allowing it to adapt quickly as applications evolve. Moreover, a mongoDB document can contain field types that other documents of the same collection do not have. Regardless of this flexibility, mongoDB still ensures expected functionalities such as full query language and consistency. [2 p. 135]

MongoDB is in the forefront of NoSQL databases, providing agility and scalability to businesses. More than half a thousand companies and start-up companies have adopted and are using MongoDB to develop new applications, refine client experience, fast track marketing time and minimize costs. [19]

4.1 Architecture

4.1.1 Document Data Representation

MongoDB stores data as document in a binary-encoded serialization called BSON or simply Binary JSON. Like in relational databases, mongoDB organizes documents that tend to have similar structure as collections. A Collection in mongoDB corresponds to a table in relational databases, a document is analogous to a row, and a field is similar to a column.

Let us consider the data model for a twitter clone application as an example. A relational database will model the data as multiple tables of User, Followers, Tweets and Retweets. However, in MongoDB the data could be represented as a single collection of Users. Each User's document might contain followers, Tweets, Retweets, represented as an embedded array. In order words, while information of a particular record in relational databases is usually spread

across multiple tables, MongoDB may have all data of a particular record in a single document.

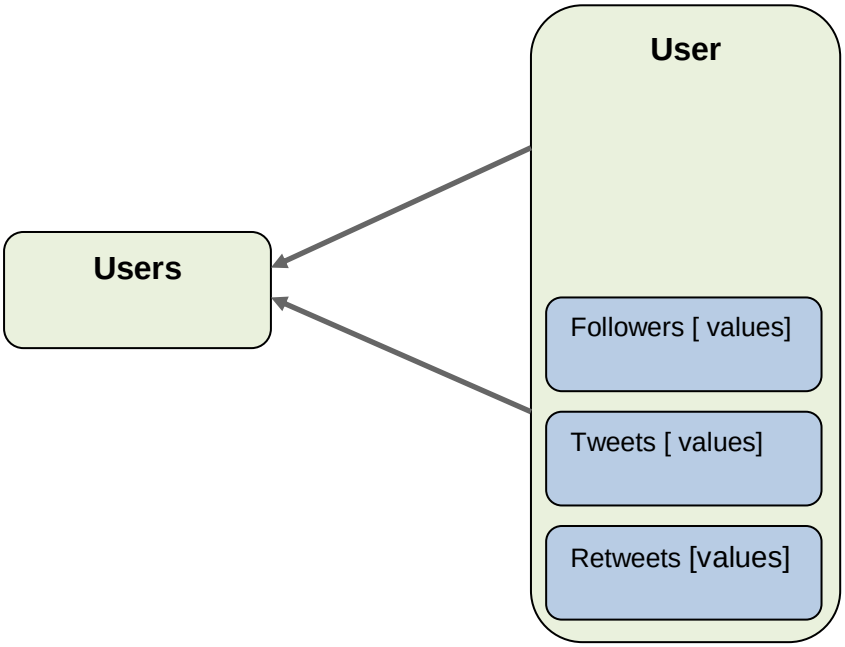

Figure 4.1. Document Data model for a Twitter clone application

4.1.2 Dynamic Schema

The structure of a MongoDB document can vary from document to document, unlike in a relational database where the structure for a row must be defined. For instance, all documents describing Twitter Users might contain user ID, tweets and followers. However, some of these documents do not necessarily have to contain user ID for one or more third-party applications. Hence, fields can be added to a document if need be, without disrupting other documents or updating the central system catalog or having system downtime. [20]

4.2 Query Model

4.2.1 Unique Drivers

MongoDB supports most well-known programming languages and frameworks by providing native drivers to enhance natural development. Each MongoDB driver is idiomatic for the respective language. Listed below are supported popular drivers:

- Ruby
- PHP
- Java
- .NET
- JavaScript
- node.js
- Python
- Scala
- Perl

4.2.2 Query Types

MongoDB queries come in different forms. A query issued to extract information from a collection may return a document or a particular set of fields within the document. Listed below are MongoDB query types.

- **MapReduce Queries** perform complex data processing expressed in JavaScript and performed across data in the database.
- **Text Search Queries** return results in the order of relevance using text arguments containing Boolean operators such as OR, AND, NOT.
- **Aggregation Framework Querries** return groups of values returned by the query, analogous to GROUP BY in SQL statements.
- **Key-value Queries** return results using a specific field in the document, usually the primary key.
- **Range Queries** return results using values defined as inequalities such as *equal to, less than, greater than, less than or equal to, greter than or equal to*.
- **Geospatial Queries** return results using proximity criteria, intersection and inclusion as specified by a point,,circle, line or polygon.

4.2.3 Indexing

Indexes in MongoDB are a significant mechanism for optimizing system performance. Inspite of the enhanced performance of operations in order of

importance, indexing poses the consequence of slower write operation, disk and memory usage. Below are listed the types of indexes MongoDB supports on any field in a document. [20]

- Unique Indexes
- Compound Indexes
- Array Indexes
- Time To Live (TTL) Indexes
- Geospatial Indexes
- Sparse Indexes
- Text Search Indexes

4.3 Data Management

4.3.1 Auto-sharding

Sharding is a technique MongoDB uses for providing horizontal scale-out for databases. It involves the spreading of data across multiple physical servers known as shards, thereby solving the hardware limitation of a single server without causing complexity in the system. MongoDB has a mechanism of balancing the data growth across the cluster and also when the cluster either multiplies or decreases. The three types of sharding supported by MongoDB are Range-based, Hash-based and Tag-aware shardings. [20]

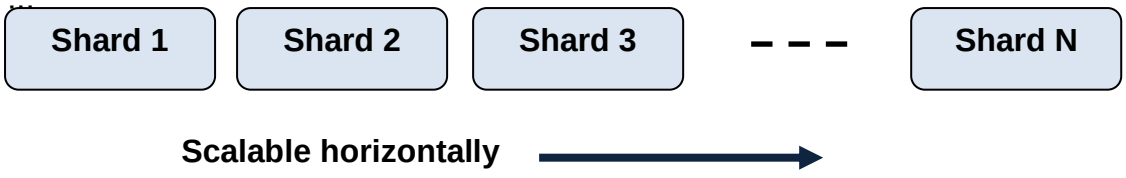

Figure 4.2. Providing horizontal scalability through sharding.

4.4 MongoDB vs. MySQL Benchmark

The benchmark was built with Python programming language and uses the latest stable drivers for each database. Drivers are the library that provide methods for connecting and executing transactions with the databases.

4.4.1 Benchmark Environment

Benchmark Machine:

- Windows OS
- AMD Turion(tm) Dual-Core CPU 2.30 GHz

Benchmark Client:

- Python 2.7

Benchmark Databases:

- MongoDB Inc. download of MongoDB
- Oracle Corporation. download of MySQL

Python Drivers:

- MongoDB *pymongo*
- MySQL *MySQLdb*

GUI for Databases:

- HeidiSQL 8.3 for MySQL
- Robomongo 0.8.3 for MongoDB

4.4.2 Test Harness

A twitter clone application, modeled in MongoDB and MySQL was used to perform Create, Read, and Delete operations. The harness permits varying the volume of benchmarks or rows to be affected, providing measurements to be analyzed and plotted as graphs. The test procedure was performed as follows.

- Author made a bulk INSERT of users.
- Author performed a READ operation to GET a user's friends.
- Author performed a DELETE operation to delete a user's friends
- The time taken to complete each of the above operations was recorded

The most crucial factor for any application is the time taken to complete a transaction. Therefore, time has been chosen as a common metric for measuring the performance of the two databases involved in this test. The test harness measures the time taken to complete an operation.

4.4.3 Database Schema

The database schema used was designed and modeled around a twitter clone application that would manage users, followers and tweets. The general schema design shown in Figure 4.3 below represents a 1:N (one-to-many) relationship between the user emtity and the other two entities.

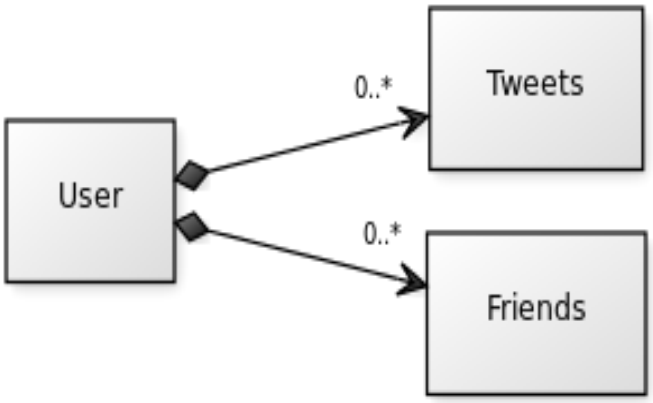

Figure 4.3. General schema design

Figure 4.4 below shows the database schema for the MySQL database implementation. It depicts a single user can tweet many times and can follow many users. To find the friends of a user, table Users is joined with table Follow, where user_id in Users is equal to user_id in Follow.

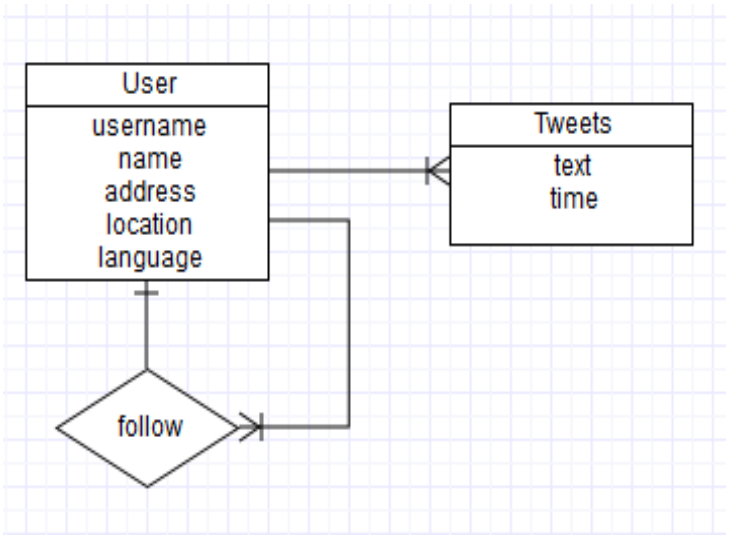

Figure 4.4 MySQL Database schema

The MongoDB database schema in Figure 4.5 embeded the Friend and Tweet documents inside the user document to maximize the advantage of sub-documents. Each user document now has its respective follower and Tweet documents nested. Likewise, Friends have embedded document containing their profiles. To find friends of a user, we will locate the user in the collection and go to his friends document where all the values can be outputed. Similarly for tweets, find the user, then go to his tweet document and output all he values.

```
{
    _id: 123456789,
    username: "danusa123",
    address: "tamk 123"),
    location: "Finland",
    language: "English",

    tweets: [
              { time: 4758869,
                text:"hello world!"}
                                    ],
    friends:[
              { username: "man123",
                address: "tamk 123"),
                location: "Finland",
                language: "English"}
                                    ]

}
```

Figure 4.5 MongoDB Twitter clone nested documents

4.5 Result and Analysis

The benchmark results analyzed here reveals how the two databases respond to both read and write operations in terms of CREATE, READ and DELETE. As mentioned earlier, the primary method of analyzing the results is through a metric, the time taken to complete an operation. The metric is plotted against the varied number of Users INSERTED, READ and DELETED.

i. INSERT 500 – 10000 users into user table and measure time.

Table 4.1 MySQL vs. MongoDB INSERT Time

Row Sizes	MySQL INSERT Time(s)	MongoDB INSERT Time (s)
500	0.348	0.003
1000	1.104	0.003
2000	0.684	0.003
3000	0.812	0.003
4000	1.230	0.003
5000	1.352	0.003
6000	1.205	0.004
7000	2.272	0.003
8000	2.547	0.004
9000	2.091	0.004
10000	3.455	0.004

Chart 4.1 MySQL vs. MongoDB INSERT Time

ii. READ 500 – 10000 users from the user table and measure time.

Table 4.2 MySQL vs. MongoDB READ Time

Row Sizes	MySQL READ Time (s)	MongoDB READ Time (s)
500	0.001	0.000
1000	0.001	0.000
2000	0.001	0.000
3000	0.001	0.000
4000	0.001	0.000
5000	0.001	0.000
6000	0.000	0.000
7000	0.001	0.000
8000	0.003	0.000
9000	0.001	0.000
10000	0.001	0.000

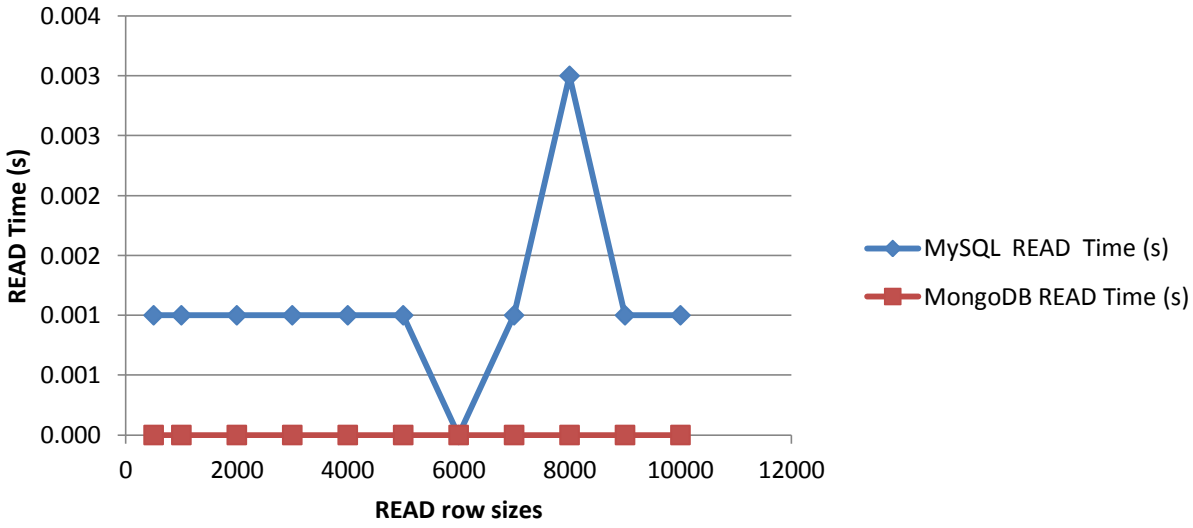

Chart 4.2 MySQL vs. MongoDB READ Time

iii. DELETE 500 – 10000 users and measure time.

Table 4.3 MySQL vs. MongoDB DELETE Time

Row Sizes	MySQL DELETE Time (s)	MongoDB DELETE Time (s)
500	0.111	0.000
1000	0.125	0.034
2000	0.144	0.001
3000	0.161	0.017
4000	0.230	0.001
5000	0.130	0.001
6000	0.241	0.000
7000	0.170	0.000
8000	0.255	0.000
9000	0.290	0.001
10000	0.253	0.000

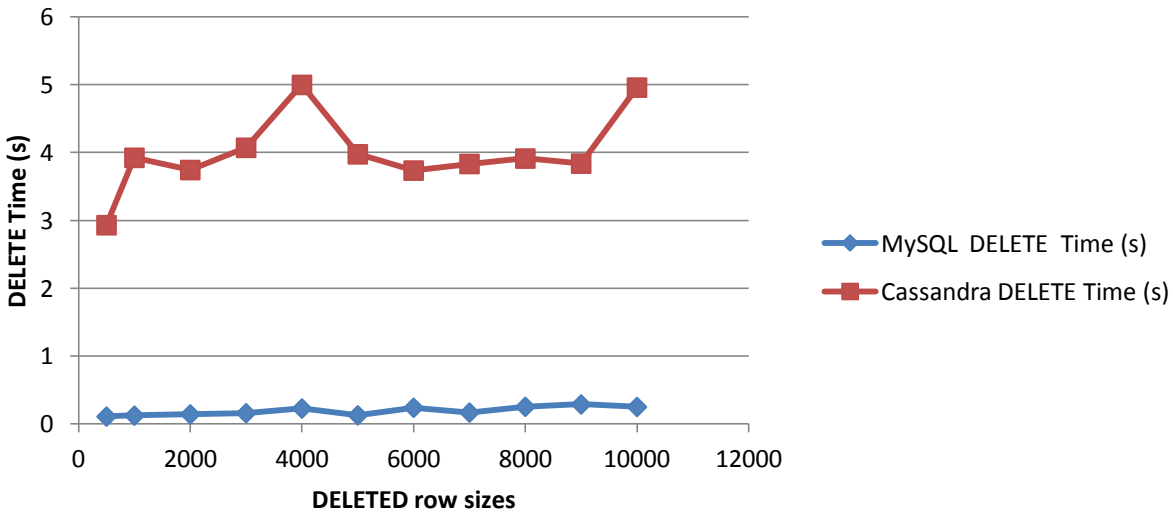

Chart 4.3 MySQL vs. MongoDB Delete Time

It can be concluded from the measurement tables and charts above that MongoDB has an overall better performance over MySQL. In all cases, MongoDB performed better in handling larger volume of data in a smaller amount of time. This becomes more evident as the number of rows inserted, read or deleted increases. MongoDB handles the different operations at a constant small time rate irrespective of the number of rows affected, unlike

MySQL which takes more time to perform operations as the number of rows grow.

5 APACHE CASSANDRA

Apache Cassandra is a column-oriented distributed database system designed to manage enormous volume of structured data spread across server cluster, while enhancing availability of service without a single point of failure. The aim of Cassandra is to operate on a framework of multiple nodes deployed across data centers in different geographical regions. It is common that components of different sizes fail at this scale, however, the persistent state of Cassandra induces the reliability and scalability of applications using this service. Although Cassandra share resemblance with traditional RDBMS in terms of design and implementation strategy, nevertheless it provides simple data model to enhance dynamic control over data structure. In addition, Cassandra was designed to take advantage of cheap commodity servers and manage high read and write output. This helps to cut cost and increase business value.

The aims of designing Cassandra has been greatly achieved. Several companies have adopted and benefited from Apache Cassandra including leading ones such as Netflix, Twitter, Cisco, eBay, Adobe and Comcast. [21]

5.1 Architecture

5.1.1 Column Data Model

In Cassandra, tables are distributed maps of many dimensions indexed by keys. Values are referred to as objects and are well structured. Row keys in a Cassandra table are strings with no limit to size. Regardless of the number of columns read or written, each operation of a single row key is atomic for each replica. A set of columns grouped together is called a column family. In addition, A column family can either be a Simple Column family or a Super column family.

Moreover, Cassandra permits applications to sort columns by either of two parameters namely time or name. Applications that take advantage of sorting column by time are those that usually display results based on time order. A good example of such applications is Inbox Search. Considering the example cited in Section 4.1.1 of previous chapter, Cassandra will model the data of the twitter clone application as shown in Figure 5.1 below. [22]

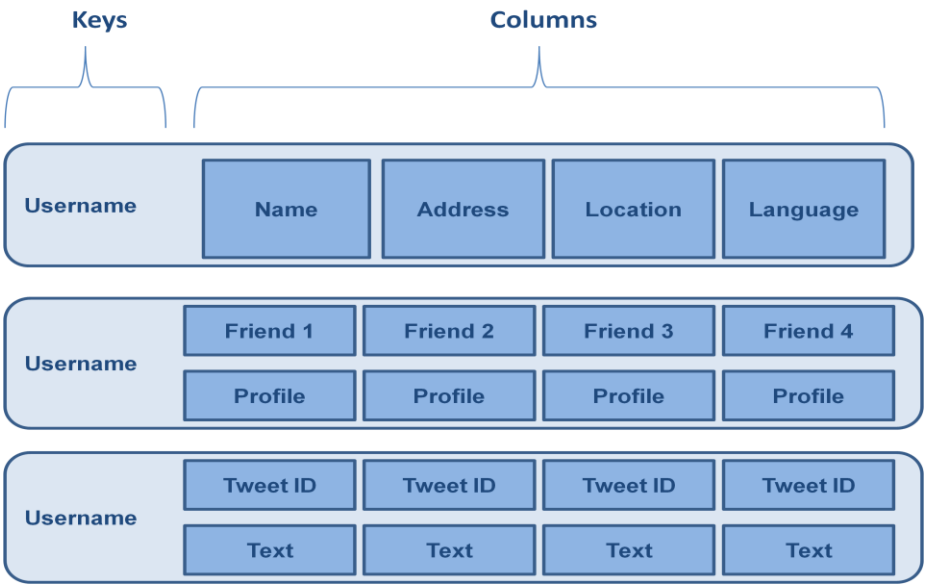

Figure 5.1 Column Data model for a Twitter clone application

5.1.2 Application Programming Interface (API)

The Cassandra Application Programming Interface include the below listed basic methods.

- INSERT(table; key; rowMutation)
- GET(table; key; columnName)
- delete(table; key; columnName)

A database system that would function well in a production environment is expected to have a complex architecture. Hence, besides Cassandra data persistency, the system should possess several other properties and core

distributed system techniques that work simultenously to handle read and write operations. Some of the these techniques are explained below.

5.1.3 Partitioning

Incremental scaling is one of the crutial design characteristics for Cassandra. This demands the ability to continuously distribute the data over the set of servers in the cluster called nodes. Cassandra uses consistent hashing as a method of partitioning data across the server cluster. This method handles the output range of a hash function as a ring. Every node in the cluster gets a random value that represents its position on the circular space. Every data with its identity key is paired with a node by hashing the identity key to produce its position on the ring. Cassandra uses this key specified by the application to channel requests. Therefore, every node becomes responsible for the area between it and its immediate neighboring node in the circle. The major merit of this method is that the going and coming of a node has no effect on other nodes except its immediate neighbors. [22]

5.1.4 Replication

Replication in Cassandra is the process the system uses to accomplish high data availability and durability. The process replicates every data item at a number equivalent to the number of hosts configured. As explained in Section 5.1.3, each identity key is given to a coordinating or master node responsible for duplicating the data items that fall within its region. Cassandra clients have variety of choices or policies of how data can be replicated. These policies determine which replicas are chosen by client applications. The popularly known policies are \Rack Aware, \Rack Unaware and \Datacenter Aware. When \Rack Unaware policy is selected by an application, the slave or non-cordinating replicas are voted by choosing all except one of the successors of the master node in the circle. In case an application chooses \rack Aware or \Datacenter Aware, Cassandra uses a system known as Zookeeper to vote a leader from its nodes. The leader is consulted by other nodes joining the cluster to determine what ranges they are replicas for.

As mentioned in previous section, nodes in the system are aware of one another and therefore the range they are in charge. Cassandra guarantees durability when nodes fail by being flexible with requirements, providing different options of replicating data. Cassandra is configured to ensure the replication of each row of data across multiple data centers. This strategy enhances managing failures of the entire data center without service interruption. [22]

5.2 Client Drivers

Cassandra like its NoSQL counterpart, MongoDB, supports common programming languages and frameworks to enhance natural development. Well-known Cassandra drivers include the following:

- Python
- Scala
- .NET/C#
- C++
- DataStax
- Ruby
- Erlang
- Go
- PHP
- Perl
- Clojure
- Node.js
- Java
- Haskell
- R(GNU S)

5.3 Cassandra vs. MySQL Benchmark

The benchmark for this experiment was built with Python programming language and uses the latest stable drivers for both databases.

5.3.1 Benchmark Environment

Benchmark Machine:

- Windows OS
- AMD Turion(tm) Dual-Core CPU 2.30 GHz

Benchmark Client:

- Python 2.7

Benchmark Databases:

- DataStax Inc. download of Cassandra
- Oracle Corporation download of MySQL

Python Drivers:

- Cassandra *pycassa*
- MySQL *MySQLdb*

GUI for Databases:

- DataStax OpsCenter v4.1.2 for Cassandra
- HeidiSQL 8.3 for MySQL

5.3.2 Test Harness

A twitter clone application modeled in Cassandra and MySQL was used to perform Create, Read and Delete operations. The harness allows varying the volume of benchmarks or number of rows to be affected, providing measurements to be recorded, analyzed and plotted as graphs. The test procedure is as follows.

- author made a bulk INSERT of users.
- Author performed a READ operation to GET all the friends of a user
- Author performed a DELETE operation to delete all the friends of a user.
- The time taken to complete each of the above operations was recorded.

As earlier mentioned in Section 4.4.2, an important factor for any application is the time it spent in completing a transaction. Hence, the test harness measures the time taken to complete an operation. The general schema design represents a one-to-many relationship between the user entity and the other two entities. See Figure 4.3 for the general schema design.

The MySQL database schema represents a user can tweet many times and can follow many users. To find a user's friend, table Users is joined with table Follow, where user_id in Users is equal to user_id in Follow. See Figure 4.4.

The Cassandra database schema is represented in Figure 5.2 below. The Users Column Family store Users, identified by a unique row key 'username'. Associated rows store user address, location and language. The Friends Column Family store Users, identified by username as the row key. Associated rows store user's friends and their profile. The Tweets Column Family store Users identified by username. Associated rows store the tweets. To find friends of a user, go to Friends Column Family, find that user by his username (the key), and pull all the associated rows (his friends). To find a user's tweets, go to Tweets Column Family, find the user by his username, and pull all the associated rows (his tweets).

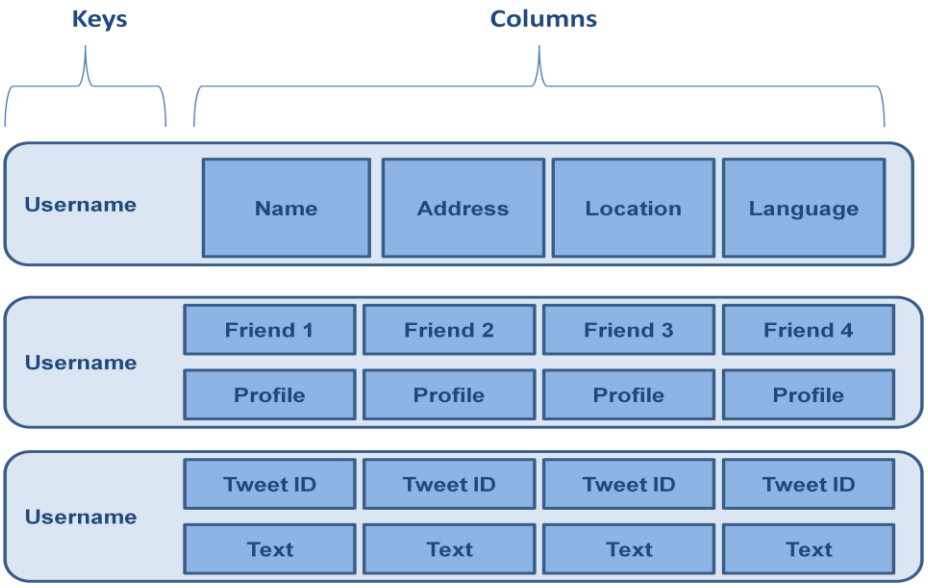

Figure 5.2 Cassandra Twitter clone Column Families

5.4 Result and Analysis

The benchmark results analysis shows how both databases involved in the test respond to read/write operations in terms of CREATE, READ and DELETE. The time metric is plotted against the varied number of Users INSERTED, READ and DELETED.

i. INSERT 500 – 10000 users into user table and measure time.

Table 5.1 MySQL vs. Cassandra INSERT Time

Row Sizes	MySQL INSERT Time(s)	Cassandra INSERT Time (s)
500	0.348	2.736
1000	1.104	2.986
2000	0.684	3.418
3000	0.812	3.841
4000	1.230	4.609
5000	1.352	5.378
6000	1.205	5.103
7000	2.272	5.651
8000	2.547	6.043
9000	2.091	6.428
10000	3.455	11.617

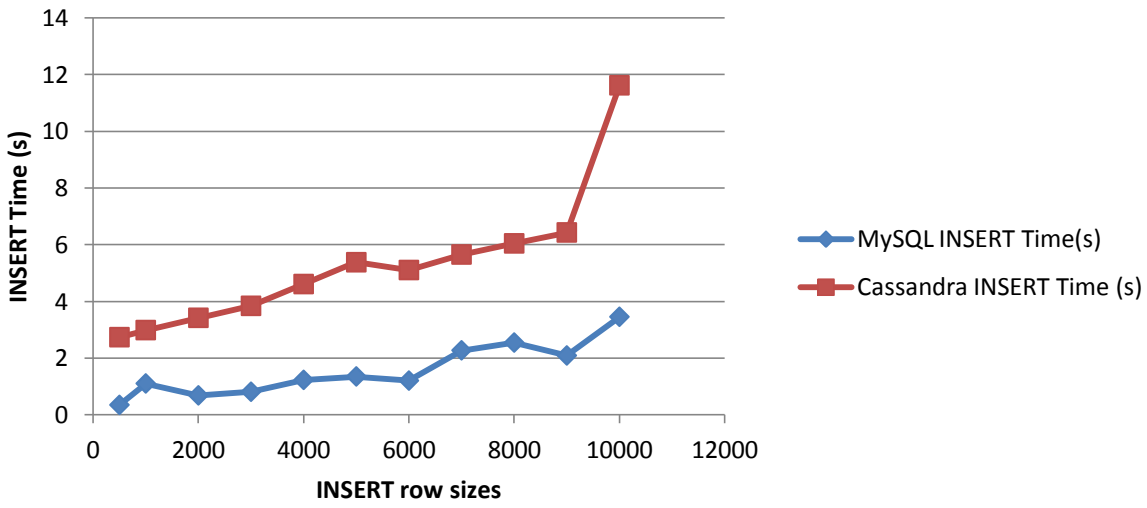

Chart 5.1 MySQL vs. Cassandra INSERT Time

ii. READ 500 – 10000 users from the user table and measure time.

Table 5.2 MySQL vs. Cassandra READ Time

Row Sizes	MySQL READ Time (s)	Cassandra READ Time (s)
500	0.001	2.671
1000	0.001	2.527
2000	0.001	2.830
3000	0.001	2.527
4000	0.001	2.532
5000	0.001	2.527
6000	0.000	2.528
7000	0.001	2.528
8000	0.003	2.574
9000	0.001	2.530
10000	0.001	2.531

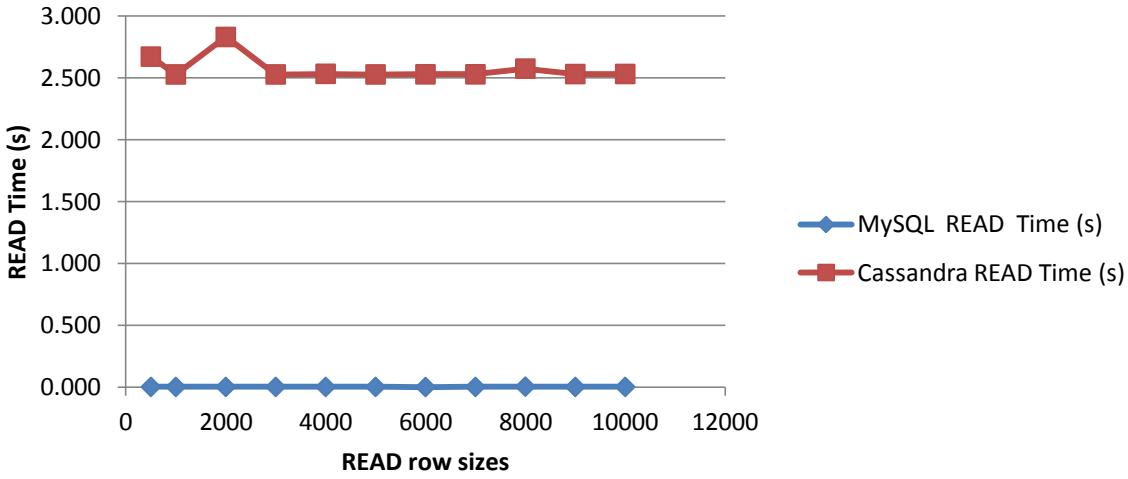

Chart 5.2 MySQL vs. Cassandra READ Time

iii. DELETE 500 – 10000 users and measure time.

Table 5.3 MySQL vs. Cassandra DELETE Time

Row Sizes	MySQL DELETE Time (s)	Cassandra DELETE Time (s)
500	0.111	2.932
1000	0.125	3.925
2000	0.144	3.745
3000	0.161	4.073
4000	0.230	5.000
5000	0.130	3.977
6000	0.241	3.737
7000	0.170	3.834
8000	0.255	3.914
9000	0.290	3.840
10000	0.253	4.960

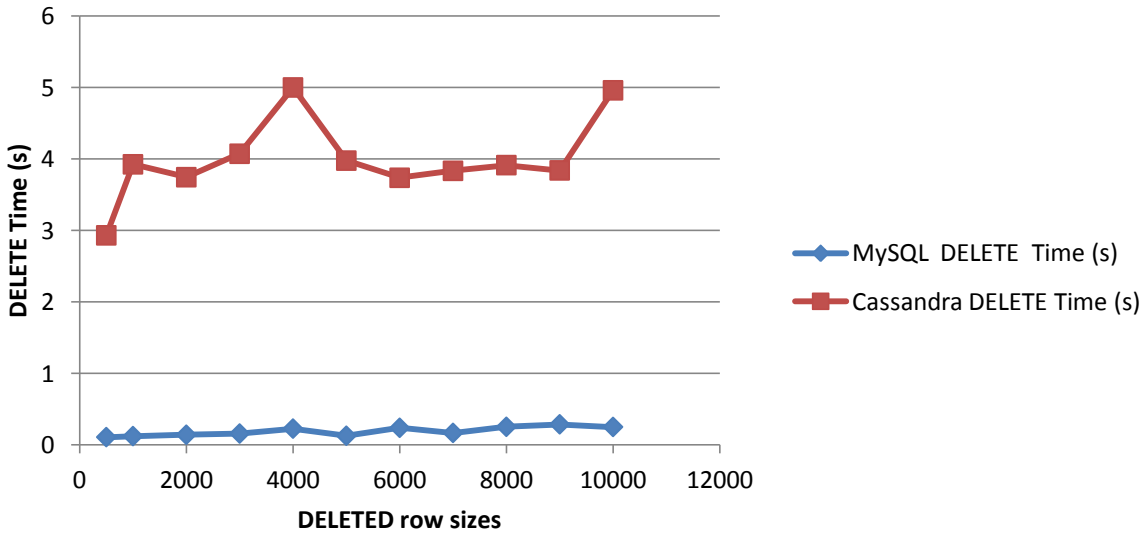

Chart 5.3 MySQL vs. Cassandra DELETE Time

It can be deduced from the recorded measurements and plotted charts that MySQL has an overall advantage over Cassandra. In all cases, MySQL has a better performance in handling larger volume of data in so little time. This

becomes obvious as the size of rows affected increases. Cassandra took longer time to complete each of the operations as the size of rows affected rises.

6 CONCLUSION

The thesis project was a research conducted to compare the scalability and user experience in terms of performance of traditional RDBMS and NoSQL databases with the intent of examining which database technology better suits which situation. Relational databases were designed to have a table structure with rows that conform to a pre-defined schema. This became their most significant characteristic as it provided logical view of the database and relations between the tables. The database schema also enhances fast and easy designing of databases and helps to remove data duplication without compromising reliability. NoSQL databases on the other hand are new and are gaining popularity for their high performance and horizontal scalability which are crutial for data centers that need enormous storage and the potential to easily expand their databases since no prior schema definition is required.

The experiments performed in the project tested and compared how each of the three databases responded to operation loads per time. The operations used in analyzing the performance and scaling abilities were INSERT, READ and DELETE operations. The time spent by each database to complete different magnitude of the operations were recorded. The measurements and charts presented in the experiment revealed MongoDB's overall performance advantage over MySQL in both read and write operations due to its simple and flexible schema, which made it cope faster with queries. However, when MySQL and Cassandra were used to model the benchmark, MySQL performed better because of its optimization to perform faster on a single node. In order words, Cassandra would perform better when scaled across multiple nodes.

In addition, the benchmark results revealed that MySQL performed considerably well in the READ and DELETE operations. This confirmed that relational databases are suitable in handling logical data operations such as simple search queries because of the logical structure.

Furthermore, the schema flexibility and subdocument advantages of MongoDB, the partitioned and reliable data store of Cassandra, are all at the expense of increasing the database size due to data duplication. Therefore, if complex query handling and scaling data across multiple nodes are essential for an application, then it is advisable to consider adopting NoSQL technology with the cost of storage and memory size in mind.

Lastly, since it is crucial for all data-driven applications to perform quite fast, database administrators and developers are required to investigate and ensure that they choose the database that best suit their application goals. This can be done by conducting a feasibility check under the settings in which the database will operate.

6.1 Further research

This research is only a small part of the entire scalability and performance research on NoSQL databases as it only examined the time spent to perform three operations at different magnitudes and used that as a yardstick to analyze the performance of each of the databases involved. This leaves room to further extend and improve the research. Research can be conducted at large scale by exploring other NoSQL databases for benchmarking. This will help database administrators and developers alike have insight into which NoSQL database best suit their project in terms of performance and also know if they should adopt NoSQL in place of relational database technology.

Besides the clues given in this project, of how well the databases handle operation loads, further research can be conducted using larger clusters to examine the scalability and performance of the databases. The complexity of the databases regarding the amount of tables, documents or columns they have may also significantly influence the performance.

Lastly, further research could also be conducted using benchmarking environments and drivers other than the ones used in this work.

REFERENCES

[1] Vaish, G. (2013) *Getting started with NoSQL*. Birmingham – Mumbai: PACKT Publishing.

[2] Carter, J. (2012) *Seven Databases in Seven Weeks.* Dallas: The Pragmatic Bookshelf.

[3] Webopedia (2014) '*Data*' [Internet]. Available from: http://www.webopedia.com/TERM/D/data.html [Accessed 25 May 2014]

[4] Wikipedia (2014) '*Data*' [Internet]. Available from: http://en.wikipedia.org/wiki/Data [Accessed 25 May 2014]

[5] University System of Georgia. (2014) A Primer on Databases and Catalogs: '*What is a database?*'[Internet]. Available from: http://www.usg.edu/galileo/skills/unit04/primer04_01.phtml [Accessed 05 May 2014]

[6] Hadjigeorgiou, C. (2013) RDBMS *vs. NoSQL: Performance and Scaling Comparison.* Unpublished M.Sc thesis. The University of Edinburgh.

[7] Wikia (2014) 'Navigational Database' [Internet]. Available from: http://databasemanagement.wikia.com/wiki/Navigational_Database [Accessed 13 May 2014]

[8] Rouse, M. (2005) *'Object-oriented database management system'*, SearchOracle [Internet]. Available from: http://searchoracle.techtarget.com/definition/object-oriented-database-management-system [Accessed 21 May 2014]

[9] Janssen, C. (2014) 'Object-oriented database', Techopedia [Internet]. available from: http://www.techopedia.com/definition/8639/object-oriented-database [Accessed 21 May 2014]

[10] Chapple, M. (2014) '*Database Normalization Basics*' [Internet]. Available from: http://databases.about.com/od/specificproducts/a/normalization.htm [Accessed 05 May 2014]

[11] Jain, A. (2013) *'Database Normalization and Normal Forms',* SQL Destination [Internet]. available from: http://sqldestination.wordpress.com/2013/01/03/database-normalization-and-normal-forms/ [Accessed 23 May 2014]

[12] Andrew, J. (2009) 'Databases'. *MongoDB Applied Design Patterns.* New York: McGraw-Hill

[13] FairCom (2014) 'ACID Properties of Transaction' [Internet]. Available from: http://www.faircom.com/ace/ace_tranacid_t.php [Accessed 06 May 2014]

[14] Hudson, A. and Hudson, P. (2013*) 'ACID Compliance in Transaction Processing to Protect Data Integrity'.* In: Helmke, M. (2014). *Ubuntu Unleashed.* Indiana: SAMS

[15] Lightwolf Technologies. (2010). *'Database ACID (Atomicity, Consistency, Isolation, Durability) Properties'* [Internet]. Available from: http://www.lightwolftech.com/index.php?page=backgrounders [Accessed 24 April 2014]

[16] Lee, D. (2014) '*The Advantages of a Relational Database Management System*', eHow [Internet]. available from: http://www.ehow.com/list_6121487_advantages-relational-database-management-system.html [Accessed 27 May 2014]

[17] Gillikin, J. (2014) '*What are the limitations of Relational Databases in business applications?*', Chron [Internet]. available from: http://smallbusiness.chron.com/limitations-relational-databases-business-applications-24159.html [Accessed 28 May 2014]

[18] Planet Cassandra (2014) '*NoSQL Databases Defined and Explained*' [Internet]. Available from: http://planetcassandra.org/what-is-nosql/ [Accessed 2nd June 2014]

[19] mongoDB (2014) '*MongoDB Overview*' [Internet]. Available from: http://www.mongodb.com/mongodb-overview [Accessed 11th June 2014]

[20] MongoDB (2014) '*MongoDB Architecture Guide*' [Internet]. Available from: http://info.mongodb.com/rs/mongodb/images/MongoDB_Architecture_Guide.pdf [Accessed 11th June 2014]

[21] PlanetCassandra (2014) '*What is apache Cassandra?*' [Internet]. Available from: http://planetcassandra.org/what-is-apache-cassandra/ [Accessed 15th June 2014]

[22] DataStax (2014) '*facebook's Cassandra paper, annotated and compared to apache Cassandra 2.0*' [Internet]. Available from: http://www.datastax.com/documentation/articles/cassandra/cassandrathenandnow.html

Wilson, C. (2010) *User Experience Re-mastered.* Burlington: Morgan Kaufmann Publishers

Oracle Database SQL Reference. 2003, '*Constraint'* Available from: http://docs.oracle.com/cd/B12037_01/server.101/b10759/clauses002.htm [Accessed 23 April 2014]

IT Business Edge, (2014). '*SQL – Structured query language'* [Internet]. Available from: http://www.webopedia.com/TERM/S/SQL.html [Accessed 23 April 2014]

Rouse, M. (2006) *'Scalability'*, SearchDataCenter [Internet]. Available from: http://searchdatacenter.techtarget.com/definition/scalability [Accessed 23 April 2014]

Nielson J. et al. *'The Definition of User Experience'*, Nielson Norman Group, [Internet]. Available from: http://www.nngroup.com/articles/definition-user-experience/ [Accessed 24 April 2014]

Mongo DB. (2014). *'NoSQL Databases Explained'* [Internet]. Available from: http://www.mongodb.com/nosql-explained [Accessed 24 April 2014]

Wikipedia. (2014). *'Apache Cassandra'* [Internet]. Available from: http://en.wikipedia.org/wiki/Apache_Cassandra [Accessed 24 April 2014]

Investopedia. (2014) Business Technology: *'Scalability'* [Internet]. Available from: http://www.investopedia.com/terms/s/scalability.asp [Accessed 24 April 2014]

Polepeddi, L. (2013) '*Relational Databases for Dummies*' [Internet]. Available from: http://code.tutsplus.com/tutorials/relational-databases-for-dummies--net-30244 [Accessed 05 May 2014]

Microsoft (2013) '*Description of the database normalization basics*' [Internet]. Available from: http://support.microsoft.com/kb/283878 [Accessed 06 May 2014]

Wrox Press (1998) 'ACID Properties' [Internet]. Available from: http://msdn.microsoft.com/en-us/library/aa480356.aspx [Accessed 06 May 2014]

ArXtecture (2014) '*Column Oriented Database-What DataBeast is this?*' [Internet]. Available from: http://arxtecture.com/column-oriented-database-what-databeast-is-this/ [Accessed 07 May 2014]

mongoDB (2014) ‘Document Databases’ [Internet]. Available from: http://www.mongodb.com/document-databases [Accessed 08 May 2014]

Couchbase (2014) ‘Scalability and performance advantgaes’ [Internet]. Available from: http://www.couchbase.com/why-nosql/nosql-database [Accessed 12 May 2014]

Copeland, R. (2000) ‘Social Networking’. *MongoDB Applied Design Patterns.* Sebastopol: O’Reilly Media

APPENDIX 1.0 Twitter Clone Application

The source codes that form the core of the project are available in the files listed below:

- main.py
- read.py
- Cassandra.py
- mongodb.py
- mysql.py

APPENDIX 1.1 (main.py)

The source code in this file empties, populate and measures the INSERT and DELETE time for each of MySQL, MongoDB and Cassandra.

```
import cassandra
import mongodb
import mysq
from datetime import datetime
import time
'''
Objective:
Test for complex queries against a social network schema
implemented across MySQL, MongoDB and Cassandra

Experiement 1: insert values into tables
Experiment 2: given a user, get his friends
Experiment 3: delete tweets posted in a particular period
'''
Result_Cass = []
Result_Mongo = []
```

```
Result_Mysql = []
Del_Result_Cass = []
Del_Result_Mongo = []
Del_Result_Mysql = []

##################################
#   Exp 1: Insert                #
##################################

#initializes all databases to 0
cassandra.empty_table()
mongodb.empty_table()
mysq.emptytable()
#modify row sizes as desired by format [500] or [500,1000]
row_sizes = [10000]
for i in row_sizes:
    print i
    person = cassandra.create_user(i)
    start = time.time()
    cassandra.insertUser(person)
    stop = time.time()
    took = stop - start
    #print "cassandra: insert and delete"
    #print took
    Result_Cass.append(took)
    start = time.time()
    cassandra.empty_table()
    stop = time.time()
    took = stop - start
    #print took
    Del_Result_Cass.append(took)
```

```
user = mongodb.create_user(i)
start = time.time()
mongodb.insert_user(user)
stop = time.time()
took = stop - start
#print "mongo db insert and delete"
#print took
Result_Mongo.append(took)
start = time.time()
mongodb.empty_table()
stop = time.time()
took = stop - start
#print took
Del_Result_Mongo.append(took)
user = mysq.create_users(i)
#
start = time.time()
mysq.insertUsers(user)
stop = time.time()
#
took = stop - start
#print "My sql insert and delete"
#print took
#
Result_Mysql.append(took)
#
start = time.time()
mysq.emptytable()
stop = time.time()
took = stop - start
#print took
```

```
    Del_Result_Mysql.append(took)
    mysq.emptyfollower()
print "------Cassandra-------------"
print Result_Cass
print Del_Result_Cass
print ("\n")
print "-------MongoDB---------------"
print Result_Mongo
print Del_Result_Mongo
print ("\n")
print "-------MySQL-----------------"
print Result_Mysql
print Del_Result_Mysql
```

APPENDIX 1.2 (read.py)

The source code in this file empties, populate and measures the READ time for each of MySQL, MongoDB and Cassandra.

```
import cassandra
import mongodb
import mysq
from datetime import datetime
import time
#initializes all databases to 0
cassandra.empty_table()
mongodb.empty_table()
mysq.emptytable()
mysq.emptyfollower()

#Fill friends/follower tables
```

```
row_sizes = [500]
for i in row_sizes:
    print i
    person = cassandra.create_user(i)
    cassandra.insertUser(person)
    user = mongodb.create_user(i)
    mongodb.insert_user(user)
    user = mysq.create_users(i)
    mysq.insertUsers(user)

##################################
#   Exp 2: Get friends of user   #
##################################

#Cassandra
print "---Cassandra---"
start = time.time()
cassandra.get_table()
stop = time.time()
print stop-start
print "\n"

#MongoDB
print "--MongoDB--"
start = time.time()
mongodb.get_table
stop = time.time()
print stop-start
print "\n"

#MySQL
```

```
print "---MySQL----"
start = time.time()
mysq.getfriends()
stop = time.time()
print stop-start
```

APPENDIX 1.3 (cassandra.py)

The source code in this file creates demo Data to populate the Cassandra database.

```
from faker import Factory
from pycassa.pool import ConnectionPool
from pycassa.columnfamily import ColumnFamily
from datetime import datetime
import uuid, random
import time
#CASSANDRA
KEY_SPACE = 'benchmark'
HOST = 'localhost:9160'
#instantiate Faker factory to create demo data to insert into Database
demo = Factory.create()
def generate_user_profile(how_many):
     profile = {}
     for i in range(0, how_many):
         userid = uuid.uuid1()
         username = demo.user_name().encode(encoding='UTF-8')
         name = demo.name().encode(encoding='UTF-8')
         address = demo.address().encode(encoding='UTF-8')
         country = demo.country().encode(encoding='UTF-8')
         language = demo.language_code().encode(encoding='UTF-8')
```

```
            profile["userid"] = str(userid)
            profile["name"] = name
            profile["address"] = address
            profile["country"] = country
            profile["language"] = language
        return profile
def generate_follower(how_many):
        follower = {}
        for i in range(0, how_many):
            username = demo.user_name().encode(encoding='UTF-8')
            name = demo.name().encode(encoding='UTF-8')
            address = demo.address().encode(encoding='UTF-8')
            country = demo.country().encode(encoding='UTF-8')
            language = demo.language_code().encode(encoding='UTF-8')
            added_at = time.time()
            #follower["follower_name"] = name
            follower["follower_address"] = address
            follower["follower_country"] = country
            follower["follower_language"] = language
            follower["added_at"] = str(added_at)
            for i in xrange(1, 500):
                follower[str(i)] = name
        return follower
def create_follower2(how_many):
        follower = {}
        for i in range(0, how_many):
            friend = generate_follower(500)
            friend_name = demo.user_name().encode(encoding='UTF-8')
            follower[friend_name] = friend
        return follower
def create_follower(how_many):
```

```
    follower = {}
    for i in range(0, how_many):
        friend = generate_follower(500)
        friend_name = demo.user_name().encode(encoding='UTF-8')
        follower[friend_name] = friend
        #print follower
    return follower
def generate_tweet(how_many):
    tweet = {}
    for i in range(0,how_many):
        user = demo.user_name().encode(encoding='UTF-8')
        text = demo.text().encode(encoding='UTF-8')
        tweet["tweet"] = text
        tweet["user"] = user
    return tweet
def create_user(how_many):
    person = {}
    for i in range(0, how_many):
        profile = generate_user_profile(1)
        username = demo.user_name().encode(encoding='UTF-8')
        person[username] = profile
    return person
def create_tweet(how_many):
    tweet = {}
    for i in range(0, how_many):
        tweets = generate_tweet(1)
        tweet_id = time.time()
        tweet[str(tweet_id)] = tweets
    return tweet
def insertUser(person):
    cpool = ConnectionPool( KEY_SPACE)
```

```
    col_family = ColumnFamily(cpool, "users")
    #print person
    col_family.batch_insert(person)
def insertFollower(follower):
    cpool = ConnectionPool( KEY_SPACE)
    col_family = ColumnFamily(cpool, "followers")
    #print follower
    col_family.batch_insert(follower)
def insertTweet(tweet):
    cpool = ConnectionPool( KEY_SPACE)
    col_family = ColumnFamily(cpool, "tweets")
    #print tweet
    col_family.batch_insert(tweet)
def empty_table():
    cpool = ConnectionPool( KEY_SPACE)
    col_family = ColumnFamily(cpool, "users")
    col_family.truncate()
def get_table():
    cpool = ConnectionPool( KEY_SPACE)
    col_family = ColumnFamily(cpool, "followers")
    col_family.get("weston94")
#get_table()
#empty_table()
#person = create_user(1)
#insertUser(person)
follower = create_follower(10)
insertFollower(follower)
tweet = create_tweet(2)
insertTweet(tweet)
#start = datetime.now()
#insertUser(person)
```

```
#stop = datetime.now()
#took = stop - start
#print took
##############Queries#################
```

APPENDIX 1.4 (mongodb.py)

The source code in this file creates demo Data to populate the MongoDB database.

```
from faker import Factory
import pymongo
from datetime import datetime
import uuid, random
import time
#MONGODB
connection = pymongo.Connection()
db = connection["benchmark"]
#users = db["users"]
#instantiate Faker factory to create demo data to insert into Database
demo = Factory.create()
def generate_tweet(how_many):
     tweet = {}
     for i in range(0,how_many):
          text = demo.text().encode(encoding='UTF-8')
          language = demo.language_code().encode(encoding='UTF-8')
          created_at = demo.unix_time()
          tweet[str(created_at)] = text
     return tweet
def generate_follower(how_many):
     follower = {}
```

```
    for i in range(0, how_many):
        name = demo.name().encode(encoding='UTF-8')
        address = demo.address().encode(encoding='UTF-8')
        country = demo.country().encode(encoding='UTF-8')
        language = demo.language_code().encode(encoding='UTF-8')
        added_at = demo.unix_time()
        follower["name"] = name
        follower["address"] = address
        follower["language"] = language
    return follower
def generate_user_profile(how_many):
    profile = {}
    for i in range(0, how_many):
        userid = uuid.uuid1()
        username = demo.user_name().encode(encoding='UTF-8')
        name = demo.name().encode(encoding='UTF-8')
        address = demo.address().encode(encoding='UTF-8')
        country = demo.country().encode(encoding='UTF-8')
        language = demo.language_code().encode(encoding='UTF-8')
        followers= generate_follower(5)
        tweets = generate_tweet(100)
        friends = {}
        for i in xrange(1, 501):
            friends[str(i)] = followers
        tws = {}
        for i in xrange(1, 501):
            tws[str(i)] = tweets
        profile["name"] = name
        profile["address"] = address
        profile["country"] = country
        profile["followers"] = friends
```

```
        profile["tweets"] = tweets
        #print profile
        #{"name":name},   {"address":address},   {"country":country},
{"language":language}
    return profile
def create_user(how_many):
    for i in range(0, how_many):
        profile = generate_user_profile(1)
        username = demo.first_name().encode(encoding='UTF-8')
        profile
    return profile
def insert_user(user):
    users = db["users"]
    users.insert(user)
def empty_table():
    db["users"].remove()
def get_table():
    db.users.find()[100]["followers"]
#get_table()
#get_table()
#for i in xrange(1, 100):
#    user = create_user(10)
#    insert_user(user)
#person = create_user(1)
#insertUser(person)
#empty_table()
#follower = create_follower(1)
#insertFollower(follower)
#tweet = create_tweet(2)
#insertTweet(tweet)
#start = time.time()
#get_table()
```

```
#stop = time.time()
#print stop-start
#took = stop - start
#print took
##############Queries#################
```

APPENDIX 1.5 (mysql.py)

The source code in this file creates demo Data to populate the MySQL database.

```
import random
import MySQLdb
from faker import Factory
import time
#MySQL
db = MySQLdb.connect("127.0.0.1", "root", "Dfasje01", 'benchmark')
cursor = db.cursor()
#instantiate Faker factory to create demo data to insert into Database
demo = Factory.create()
def create_table_users():
    cursor.execute("""CREATE TABLE users (
    id serial AUTO_INCREMENT PRIMARY KEY,
    name varchar(100) NOT NULL,
    address varchar(200) NOT NULL,
    country varchar(100) NOT NULL,
    language varchar(10) NOT NULL) """)
    db.commit()
def create_table_followers():
    cursor = db.cursor()
    cursor.execute('''CREATE TABLE followers (
```

```
    id serial AUTO_INCREMENT PRIMARY KEY,
    user integer NOT NULL REFERENCES users.id,
    follower integer NOT NULL REFERENCES users.id)''' )
    db.commit()
def create_table_tweets():
    cursor.execute('''CREATE TABLE tweets (
    tweet_id serial AUTO_INCREMENT PRIMARY KEY,
    user_id integer NOT NULL REFERENCES users.id,
    text varchar(300) NOT NULL,
    date TIMESTAMP)''')
    db.commit()
def generate_user_profile(how_many):
     profile = {}
     for i in range(0, how_many):
         username = demo.user_name().encode(encoding='UTF-8')
         name = demo.name().encode(encoding='UTF-8')
         address = demo.address().encode(encoding='UTF-8')
         country = demo.country().encode(encoding='UTF-8')
         language = demo.language_code().encode(encoding='UTF-8')
         person = [name,address,country,language]
         profile[i] = person
     return profile
def generate_tweet(how_many):
     tweets = {}
     for i in range(0, how_many):
         tweet = demo.text().encode(encoding='UTF-8')
         tweets[i+1] = tweet
     return tweets
#create_table_users()
#create_table_followers()
#create_table_tweets()
```

```
def create_users(n):
    user = {}
    for i in xrange(1, n+1):
        users = generate_user_profile(1)
        for key, value in users.iteritems():
            for el in value:
                el.split(",")
                name = value[0]
                address = value[1]
                country = value[2]
                language = value[3]
        user[i] = [name,address,country,language]
                #user.extend(name, address, country, language)
    return user
def insertUsers(user):
    for k, v in user.iteritems():
        data = [v[0],v[1],v[2],v[3]]
        cursor.execute("INSERT into users(name, address, country,
language) values(%s, %s, %s, %s)", data)
    db.commit()
def create_followers(n):
    for i in xrange(1, n):
        data =[i, n+2, i]
        cursor.execute("INSERT INTO followers VALUES (%s, %s,
%s)",data )
            #cursor.execute("INSERT into users(name, address, country,
language) values(%s, %s, %s, %s)", user1)
    db.commit()
def create_tweets(n):
    for i in xrange(1, n):
        tweet = generate_tweet(n)
        for key, value in tweet.iteritems():
```

```
            data = [102, value]
        cursor.execute("INSERT into tweets(user_id, text) values(%s,
%s)", data)
        #print data
    db.commit()
def emptytable():
    cursor.execute("DELETE from users")
    db.commit()


def getfriends():
    cursor.execute("""SELECT follower, users.*
                      FROM followers, users
                      where user = 502""")
    db.commit()
def emptyfollower():
    cursor.execute("DELETE from followers")
    db.commit()
#d = create_users(1000)
#insertUsers(d)
#create_tweets(500)
#create_followers(500)
```